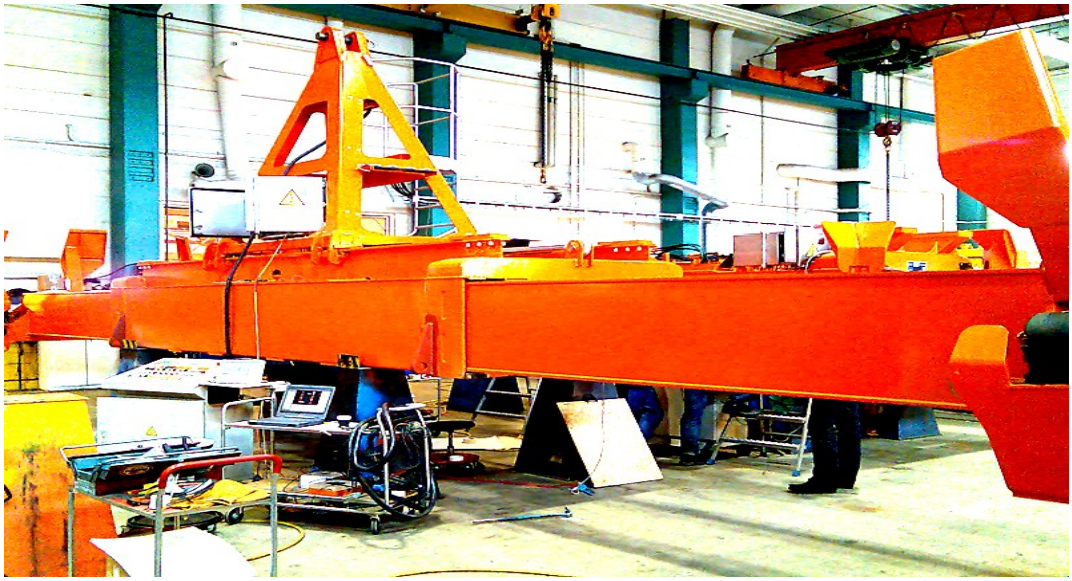


CoDeSys on CCP XS for Bromma Conquip

Joel Ek winter 05/06

Master Thesis 20p, D-level, Department of Computer Science and Electronics
Mälardalen University

Supervisor: Johan Stärner, Examiner: Hans Hanson

ABSTRACT

The objective with this thesis has been to investigate the possibility to base a control system steering container spreaders with an on-board computer and a software Programmable Logic Controller (PLC). The aim is that CoDeSys, a software environment that follows the standard for PLC programming, is to be used. The PLC is then to be programmed with this tool and the onboard computer can then act as control unit for the spreader.
In the current system, the logic for the process is build by a number of software components plus inputs and outputs that is connected in an application program.
The idea is that the components are to be altered to make them function with CoDeSys, and that we then can use a graphical language in CoDeSys to connect components and signals.

The result of the thesis shows that the concept holds, the porting of a component from the current system to one based on the on-board computer CCP XS, developed by CC-Systems, have been performed and prove to be possible. The performance test that has been made on the chosen ported component show that that change of the control system to this new generation will give a considerable performance improvement. Besides the performance improvement the other reason for changing the system is the large display on the CCP XS, it can be utilized for visualizations of logs and diagnostics with CoDeSys provided tools.

Keywords: CoDeSys, PLC, CCP XS, Spreader control system, on-board computer

SAMMANFATTNING

Uppgiften med detta arbete har varit att undersöka möjligheten att basera ett styrsystem för lyftok på en fordonsdator innehållande en Programmable Logic Controller[1] (PLC). Tanken är att CoDeSys, ett utvecklingsprogram som följer standarden för PLC programmering ska användas. PLC:n ska på så sätt programmeras och fordonsdatorn kan sedan agera som styrenhet för hela oket.
I nuvarande system byggs logiken för styrningen upp av ett antal mjukvarukomponenter samt in och utgångar, dessa kopplas samman i ett applikationsprogram.
Idén är att komponenterna ska ändras för att kunna användas i CoDeSys och att vi sedan kan använda ett grafiskt språk i CoDeSys för att koppla ihop porterade komponenter och signaler.
Resultatet av arbetet visar att idén håller, en portering av komponenterna från nuvarande system till ett baserat på fordonsdatorn CCP XS som utvecklas av CC-Systems har gjorts och visats sig möjligt. Prestandajämnförelsen som gjorts på porterad komponent visar också, som förväntats, att en övergång till denna nya generations styrsystem för lyftok ger en avsevärd prestandaförbättring. Den andra orsaken till en förändring av systemet är skärmen på XS:n. Den kan utnyttjas för visualisering av loggar och diagnostik med CoDeSys inbyggda verktyg.

Nyckelord: CoDeSys, PLC, CCP XS, Kontroll system för lyftok, fordonsdator

[1] Översättning svenska: programmerbar logisk enhet

CONTENT

.....1

1 Background.....6

1.1 Problem formulation.....7

1.2 Goal and purpose.....8

1.3 Delimitations.....8

1.4 Disposition.....8

2 Related work.....9

Preliminary studies.....12

2.1 SCS2.....12

2.1.1 ABE.....14

2.1.2 BMS.....15

2.2 CoDeSys.....16

2.2.1 The IEC 61131 standard.....16

2.2.2 Languages.....16

2.2.3 Program Organization Unit.....17

2.2.4 Log and error handling.....17

2.2.5 CoDeSys SP.....18

3 Problem analysis.....20

3.1 Porting of components.....21

3.1.1 The setting of component parameters.....23

3.1.2 Change number of in/out- puts.....23

3.2 Converting the Application description.....23

3.3 Replacing system for logging and diagnostic.....24

4 Method.....25

4.1 Porting of components.....25

4.1.1 Selecting POU to use for the components.....25

4.1.2 Selecting language for the POUs.....25

4.1.3 Selecting language for component board application.....26

4.1.4 The setting of component parameters.....27

4.1.5 Change number of in/out- puts.....28

4.2 Converting the Application description.....29

4.3 Replacing system for logging and diagnostic.....29

5 Solution.....30

5.1 Porting of components....30
5.1.1 Function Block with External library....30
5.1.2 Implementing the library functions....31
5.1.3 Porting of the components C++ code into ANSI C....31
5.1.4 Changing component parameters....35
5.1.5 Change number of in/out- puts....36
5.1.6 Retain variables....36
5.2 Converting the Application description....37
5.2.1 Name of Converter Demonstrator....38
5.2.2 Pseudo code for the converting tool....38
5.3 Replacing system for logging and diagnostic....41
5.4 Conclusion of solution....42

6 Testing....43
6.1 Performance Test....44
6.2 Size of ported code....45
6.3 Efficiency of produced code....46

7 Discussion and Conclusions (Survey)....47

8 References....48

Appendix 1 Quick start guide....50

Appendix 2 Terms och definitions....51

Appendix 3 CoDeSys file in XML representation....52

Preface

I would like to thank CC-Systems for offering me the opportunity of performing my thesis there and of course give special thanks to my supervisor Fredrik Löwenhielm.

Thanks also to Mälardalen University from who I owe my education, especially Johan Stärner, my supervisor at the school who helped me through this thesis, the final step in my education.

1 Background

CC Systems (CCS) is a company founded in 1991, the growth has been good and the company has today 130 co-workers at offices in Alfta, Uppsala, Västerås, Örnsköldsvik and Tammerfors. The company works with production of electronic and software with specialization in development of advance information and control systems for machines that work in a demanding environment.

Examples of machines can be forestry machines, military vehicle but also a spreader for containers. The control system on a spreader must both have high reliability and be robust when it is manoeuvred under extreme conditions; it should handle being exposed to tropical and arctic climate, saltwater and chock loads.
CCS has already developed a system on Bromma Conquips spreaders today. But they are now in a process of developing the next generation control-system for their spreaders with a control monitor mounted directly on the spreader. It is possible that CC Systems on-board computer CCP XS can be used in this new system and work as both master[2] and display (fig. 1).

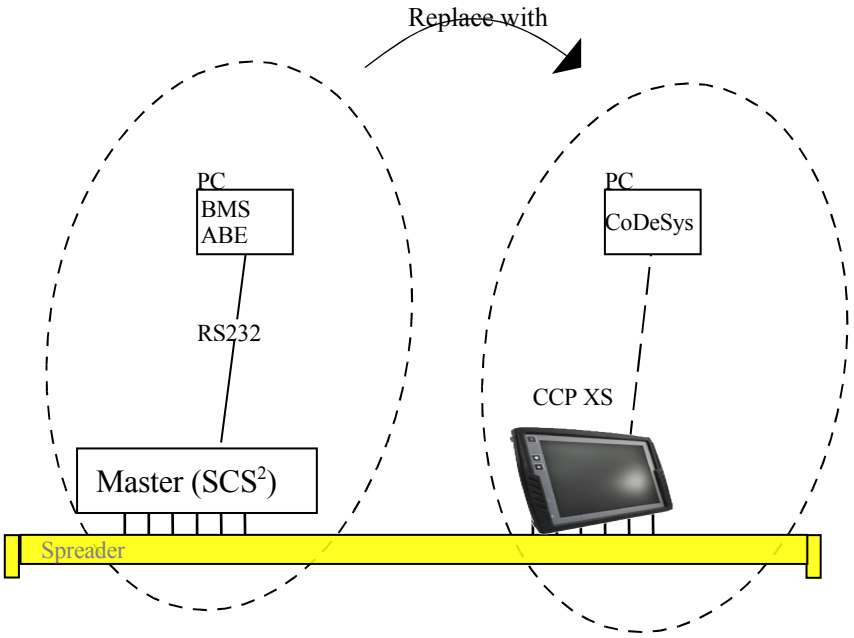


Figure 1: Showing the main idea being evaluated in the thesis

CCS use the software CoDeSys which is based on IEC61131-3 to program control-systems for the CCP XS, the question is how to best reuse and port parts from the old control-system to use in the new [1].

[2] The master collects input from the other computer nodes in the system, performs calculation and send out new outputs

1.1 Problem formulation

The aim with this thesis project is to evaluate how Bromma Conquips current control-system for spreader can be replaced with a control-system based on CCP XS and CoDeSys. A specific software component is to be ported from the current system to the new and performance between the systems will be evaluated. A software component could for example be the logical AND gate, it consists of two parts: the interface and the logic. The interface describes how a component correlate to the system, it could be other components or for example signals direct from sensors. The logic describes how the input to the component is to be presented as output signals. If the AND gate has two inputs and receives a signal at both of them at the same time then the software logic tells him to present a signal at the outgoing port. All components function in the same way, reading inputs and calculating the outputs according to their logic.

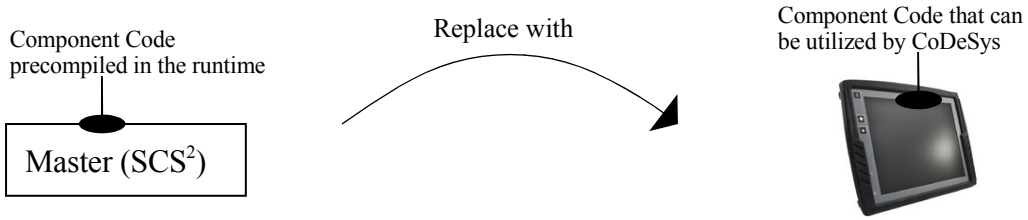


Figure 2: Illustrating the problem of changing the implementation of the logic

The porting consists of two parts, one part is the code that implements the software- components and their logical function (fig.2), the other part is the application code deciding how the different software part correlate to each other (fig.3).
An evaluation will be done for both these parts to see the possibilities to replace or reuse them: The software-components must in some way be ported so they can be used with CoDeSys and the application-description need to be translated so that CoDeSys understands it.

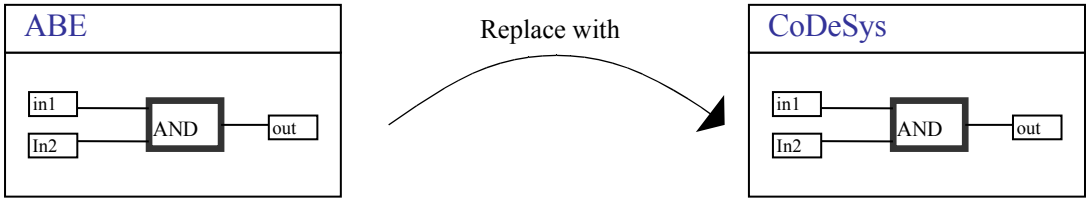


Figure 3: The building of applications is to be taken over by CoDeSys

The other part of the thesis is to evaluate how Bromma Conquips current system for logging and diagnostic can be replaced. The software program managing this is running on an externally connected Personal Computer (PC), it is connected to the system when needed. CoDeSys provide a tool for logging that hopefully could be used to replace the connection of an extern PC [2].

1.2 Goal and purpose

The goal is to finally have a functioning demonstrator with a demo-application that uses ported parts from Bromma Conquips software on the CCP XS together with CoDeSys. A specific selected component is expected to be ported with a successful outcome. If so the CCP XS, together with the ported component and CoDeSys, is going to be tested and used to control a spreader.

1.3 Delimitations

The porting will only be made on a few components, this include the conversion of the application description that also only will function for those components.
One specific component, one of the most complex will also be chosen and tested at site in one of Bromma Conquips factories. The test will prove that the concept works and receive performance data.
The method for porting the other components is the same and if variations occur they are discussed later in this thesis report, the workload is probably also enough without having to port all the Bromma specific components.

1.4 Disposition

I start by giving some preliminary knowledge about CoDeSys and how Bromma Conquips control system work today, and then also discuss what is to be changed and the reasons.

Then an analysis is made with identifying of different investigation parts for the thesis.
After this a discussion of methods solving the different parts is made.
Finally the solution used to solve the problems for the various parts is presented.

After this the result from the test is presented and analysed follow by discussion and conclusion.

2 Related work

This thesis is quite strict in the specifications for how the work is to be performed, making it a unique investigation where similar work are few, one reflection of this is of course that the thesis in some sense would be superfluous if this was not the case. The reason for this is the tool CoDeSys that has been chosen as the standard tool developing new control systems at CC-Systems. A thesis performed earlier at the company evaluated different kinds of control programming tools that followed the IEC[3] 61131-3 standard. It came to the conclusion that CoDeSys was most suited for the kind of applications that was needed in the business the company worked towards. For deeper information about CoDeSys and its advantages over similar tools can be found at the following reference [3].

In a broader sense the central concern in the thesis is a porting of software components to make them work in a new context. Software portability is one of the topics in Software computer science that has involved most research, especially in how to produce the software in an independence way.
James D. Mooney defined portability to be [4]:

"An application is portable across a class of environments to the degree that the effort required to transport and adapt it to a new environment in the class is less than the effort of redevelopment."

The portability can be divided in two steps: a physical, moving the code to the new environment, and an adaptation step modifying the code for it to work in the new environment. Two common ways of porting a program is binary, where the executable program is ported, and source portability where the code is ported to the new environment.
Binary porting is the harder to achieve because the environments (processor and operative system) have to resemble each other a lot.
Source porting is easier to perform but leads to some additional steps (compiling and linking) before an executable program can run in the new environment.
Mooney states that the problems porting a program reside from the program interacting with its environments interfaces (processor, memory, runtime support procedures and operating system). The solution is to keep a consistent structure for each environment and adapt the program to handle inconstancies that still remain [4].

In the paper "Approaches to Making Software Porting More Productive", Toshikiyo Tanaka, Mitsuari Hakuta, Norikazu Iwata and Masato Ohminami discuss two general ways of making porting more effective. The first way they say is to avoid barriers for porting directly at design stage. This can be guidelines to handle interface specification barriers such as processor dependant use of alignment and padding, libraries using processor dependent functions or system calls specifications.
Then isolate and give porting instructions for those cases in the program where a modification later is necessary.

[3] International Electrotechnical Commission: for information about the 61131-3 standard see section 3.2.1

The second way to heighten the effectiveness is to collect all available knowledge about porting and also use the tools available. Ten different guidelines and tools are presented and the uses of them are estimated to decrease the workload in man-hours by 40%. The tools described are tools for: porting compatibility checking, portability evaluation, generating system calling routines, program structure viewing, OS emulator and test support [5].

In "Strategies for Porting Software between Microsoft Windows and Apple MacOs" William Edward Woody describes the process of porting applications between two of the most common operating system platforms. In his paper he outlines three main strategies when porting software from Microsoft Windows and Apple Macintosh:

1. Factoring the user interface: A strategy suitable for applications that do not rely on underlying Windows API[4] calls. Steps that need to be taken are: find the interfacing to user interface, port the code and then recreate the user interface in Macintosh. If the user interface is a large part of the application this strategy is not cost effective.

2. CView[5] translation Layer: A Strategy for applications that rely on the user interface (for example the Microsoft Foundation classes (MFC)). Steps that need to be taken are: find the interfacing to user interface and CView, port the code, create a "translation layer" that converts the MFC events and methods to the target framework counterpart and then recreate the user interface in Macintosh.

3. API Translation Layer: A Strategy for applications that use the Win32[6] API. Steps that need to be taken are: find the interfacing to Win32 API, port the code, create a "translation layer" that converts the Win32 API events and methods to the target framework counterpart and then recreate the user interface in Macintosh.

To find the best strategy for each application can also be a problem and many times there is not only one strategy that solves the whole porting, instead a mix of different approaches is necessary [6].

The research on increasing the software portability made during the years concludes in the following strategies: control the interfaces, isolate dependencies and to think portable.
The measurement for when to port existing code instead of redevelop is an estimation of cost. When software porting has been chosen as the best alternative a porting document is ideally created, the implementation adaptation should be kept to a limited subset of the program, this decrease the possibility to introduce new errors in the code. The program still needs to be tested to see if the entrance to new environment itself has resulted in new errors [7].

In our case the effort for porting in stead of redevelop is predicted, primary for the direct cost being lower when porting the code but also reason that additional costs for testing the software will be smaller.

[4] Application Programmatic Interface, provides services for computer programs.

[5] CView is a MFC base class providing functionality for user defined view classes

[6] Win32 is an API to program in windows, it mainly consist of C –code functions

A related topic that sometimes is mixed up with porting is reuse. The similarities are many but there is a distinguished difference between the two. With porting the aim is to change the environment for the whole application, with software reuse an individual part, a component is developed that can be reused in various applications [7].
The component is developed in a certain "framework", the role of the framework is to abstract the interfacing environment and divide the parts interesting for the development in a fixed and variable part. The fixed part can be used by developers but not modified, it makes a produced component function in a broad range of environments where the framework is used.
The variable part is then used to make useful individual components [8].
The most famous technique in this direction is developed by Microsoft, the .NET framework API [9].
But until Microsoft has taken control of the entire world of computing the fact remains that software porting and reuse is an important field of research. Several organisations have made special groups dealing only with these topics and hold annual conferences on the matter [10, 11].

PRELIMINARY STUDIES

In order to get an understanding of how to carry out the thesis I required some background information. From Bromma Conquip AB, the company for which the thesis is performed towards I needed information about the system they are using today. CCS developed this system so this was of course not a problem.
A better understanding of CoDeSys was also essential when this tool was completely new to me. So the preliminary studies have been performed by reading documents and user manuals for the systems. Tests of the different software tools, both those used in Bromma Conquips system and CoDeSys have also been conducted.

2.1 SCS²

The second generation of the Spreader Communication System (SCS²) is the core in a Bromma Conquip spreader. The hardware controlling the spreader is divided into several nodes and resides on both the spreader and the crane (fig.4), the number of nodes used depend on the spreader. At least one is required, the Master that makes all the calculations and decisions for the spreader, it is located on the actual spreader. Then you can connect up to seven other nodes on the spreader and up to four nodes on the crane. These slave[7] nodes act as remote I/O[8] to the master.

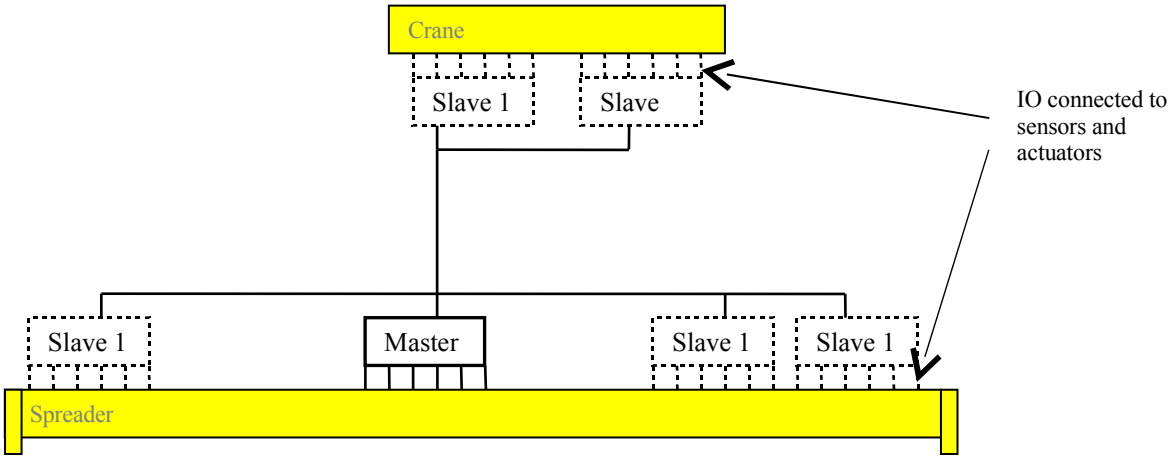


Figure 4: Graphical overview of the SCS² system

The behaviour of the system is described by a number of software components and connections. The nodes on the spreader have I/O directly connected to the sensors and actuators on the spreader.
A crane node could be interfacing the control system or a monitoring screen.

The communication between the nodes is made through a Controller Area Network (CAN) network. It is a multi master network that connects several electrical units. A message send on

[7] The slave sends its input values to the Master and then set output values according to received data from the Master.

[8] I/O: Input and Output

the network is read my all units, but only the unit that that the message is intended for process the message [12].

The software in SCS[2] is divided in three parts:

- Boot Program: The Boot program runs when the system is switched on and is responsible for things as initialising hardware and performing program download.

- System Program: After the initialising part the System program starts running and continues until the system is turned off. It is responsible for interpreting the Spreader program, supporting connected BMS [9]and logging of events in the event log. The system program also performs the reading and writing of all input and output signals.

- Spreader Program: The third part of the system is the Spreader program, it is created in a software tool called ABE[10]. This program describes the logic of the spreader, how the different components in the system are connected. It is not executing by itself, instead it is read and used by the System program. The spreader program is updated with the download function in the Boot program [13].

[9] See section 3.1.2

[10] See section 3.1.1

2.1.1 ABE

The Application Builder Environment (ABE) software tool is used to create the spreader programs. The spreader program describes the functionality of the nodes by deciding which software components should run on the node and describing the connection both between different components but also between components and input/outputs.
ABE has a graphical interface (fig.5) which makes it easy for the user to create the logic for the SCS[2].

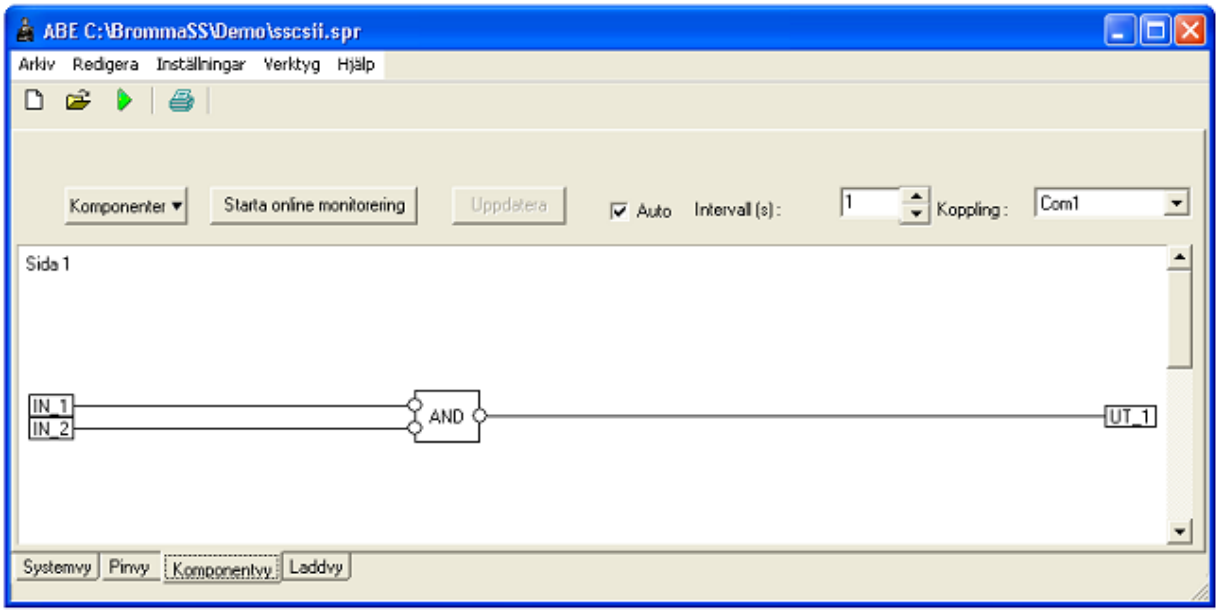


Figure 5: A simple component board, an AND component is connected to two in-ports and one out-port.

To build the component board[11] the user selects the components he wishes to use from a list and put them on the board. Then on each port of a component he can choose to which other component port it should connect. If he whish to connect it directly to the nodes I/O he first need to activate the specific I/O port by giving it a name.
With double-clicking the left mouse button over the component the user can also access the parameter setting of the component and alter them if he wishes.

When the user or the programmer of the system has decided which components to use and how to connect them he saves the program and a spreader file is generated. This file does not hold any information about the logic of the individual components instead it gives information about which nodes, components, connections and ports that are to be used.
All components and their functionalities do already exist within the target system as precompiled objects.
When a PC connect and the file is downloaded, the System program reads the spreader file and builds up a component board containing all the components, connections and ports as described in the file [14].

[11] Component board will be used as term to describe a set of components and ports and their interconnections to each other.

The structure of the file must in order for the system program to interpret it follow a specific format according to the figure below:

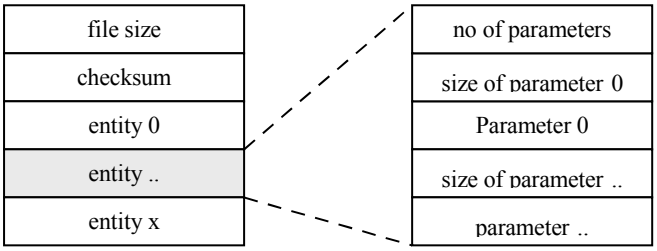


Figure 6: Showing the structure of the spreader-file generated in ABE

Four different types of entity exist; general information, system information, components and connections.

- General information: Each spreader program file holds one entity with general information describing things as time of creation, name of creator and type of spreader.

- System information: There should also be precisely one entity with system information; it holds system specific information such as number of nodes in the system.

- Components: Each component used in ABE will be described with an own component entity in the file. It holds information about type and name of the component, but also component specific parameter settings.

- Connections: After that all components have been defined the connection entities are listed. Each of these entities contains the information: source component name, source component port name, destination component name and destination component port name [15].

2.1.2 BMS

Bromma Conquips control system for the spreaders is called Bromma Monitoring System (BMS).
BMS is a software program that is connected to the system at need, the nodes on the spreader have one port where a PC containing BMS can connect.
When BMS is connected to a node it can be used to:

- Download the application description or the entire system program
- Retrieve the log messages
- Check the status of the I/O ports of the node
- Force values to ports.

BMS is also used to monitor and set general information of the spreader, it can be things a time & date and version information of different software on the spreader [16].

2.2 CoDeSys

Controlled Developed System (CoDeSys) is developed by Smart Software Solutions (3S) and they describe their software as a complete development environment for your PLC.
CoDeSys implements the standard IEC 61131-3 and 3S is today one the leading manufacture in programming PLC systems.

2.2.1 The IEC 61131 standard

The IEC 61131-3 standard is a standard for programming languages that is used in industrial automation, typically to program a PLC. The standard consists of two parts: Common objects and Programming languages [17, 18].

2.2.1.1 Common objects

The common elements define which data types that must exist, how the variables should be declared (input, output and global etc). It also defines that a PLC program is to be build using Programming Organising Units (POU), these POUs are: Program, Function Block and Function.

2.2.1.2 Programming languages

The other part defines which languages that should exist in order to follow the standard. They are: Instruction list, Structured Text, Function Block Diagram, Ladder Diagram and Sequence Function Chart.

2.2.2 Languages

CoDeSys support all the standard languages defined for PLC programming and implements one more, the Continues function chart.
Description of the different language in CoDeSys:

- Instruction List (IL) is a text based language similar to assembler language.
- Structured Text (ST) is a text based higher layer programming language.
- Function Block Diagram (FBD) allow the user to graphically connect different function blocks, both standardized functions like AND but also own user defined. And in that way create a more complex function.
- Ladder Diagram (LD) is the correspondence to ladder logic in the IEC 61131-3 standard and is a graphical language to connect relay and coil.
- Sequential Function Chart (SFC) is used for programs that can be divided into step, each step contains some action and the transition to the next step is controlled by a logic condition.
- Continues Function Chart (CFC) is similar to FBD, not as fast but more flexible, allowing both continuous connections and change of execution order.

2.2.3 Program Organization Unit

In CoDeSys you can create three different types of Program Organization Units (POU); program, function blocks and functions. The different POUs can be written in one of the 6 languages supported. Explanation of the different POUs:

- Program: A Program is recognised globally in the project, it can return several values at execution and all values in the program are retained between different executions. It is not possible to make different instances of a program.

- Function Block: With Function Block it is possible to make instances[12] where each instance will have a unique name and hold its own structure of variables. It can have several input and output variables, these can be accessed by other POUs whereas the internal variables can not.

- Function: A Function can have several inputs but only one output variable. It is not possible to make instances of Functions and a specific input to a Function will always lead to the same output.

2.2.4 Log and error handling

2.2.4.1 Log

CoDeSys provide a tool called Log, it stores all actions that is performed in online mode and is used for monitoring and trace errors in the program.
It is possible to open the log afterward in offline mode and save extracts from the log in external files. The whole Log is during the online session saved in a binary file.
The type of action that is saved in the log is chosen before and is of the following categories:

- User action: performed by the user and can for instance be ‘Force values’.
- Internal action:
- Status change: The runtime system has changed status.
- Exception: An exception has happened.

Each action in the log is described with: type of category, description of action, information about errors occurred during the action, system time, time since start of online session and finally the duration of the action.

2.2.4.2 Alarm configuration

Alarm configuration is another tool included in CoDeSys, it is used to organize alarm event in a program. These alarms can be errors or other critical process states that the user needs to be notified about. The occurrence of an alarm event can be both visualized and saved to a history file.

[12] Copies of the same dataset often used in object oriented programming

The alarm configuration consists of two parts: Alarm Class and Alarm Group.

2.2.4.2.1 Alarm Class

The Alarm class works as a general class to the alarm group and decide base configuration such as, how to handle acknowledgements and what action to perform when a specific alarm is detected.

The different actions that can be performed at the occurrence of an alarm event are:
- Save: Internally saved in a log file.
- Print: A message is printed.
- Message: A message will be displayed
- E-Mail: A message will be send to an e-mail address.
- Variable: A variable in CoDeSys will be assigned the alarm status and a message.
- Execution: An executable file will start.

The above mentioned message can be defined by a number of predefined information types: Date, time, priority, message as defined in the specific alarm among others.

2.2.4.2.2 Alarm group

All alarms belong to a specific Alarm group, the common denominator in the group is a deactivation variable plus saving options for the Alarms in the group. But it is possible for the alarms in the group to belong to different Alarm classes.

When you add a new alarm to a group the following parameters can be set:
- Expression: Specify the variable or expression that should trigger the alarm.
- Type: Regards to what value the variable should trigger
- Class: Decide what Alarm class it should belong to.
- Priority: Priority level 0-152.
- Message: The text that will be showed in case the alarm is triggered.

If a save action has been selected for a class you must specify in the alarm group where to save the alarm information. This file will hold the alarm information that is saved for all alarms in the group [19].

2.2.5 CoDeSys SP

3S provide a softPLC, basically a software program that acts like a runtime system for the developing tool CoDeSys. The runtime was developed to function on many different operating systems and was therefore programmed in ANSI[13] C. The softPLC is programmed by a programming tool, in this case CoDeSys [20].
It is also with these runtimes that 3S get income; the developing tool is free of charge but need a runtime in order to be of use. Each device must buy a runtime license [21].

[13] American National Standards Institute: A non-profit standardization organization

In our case the softPLC runtime will be running under Windows CE[14] on the on-board computer CCP XS.

[14] A smaller variant of Microsoft's operating system Windows, primary used in embedded systems

3 PROBLEM ANALYSIS

CC Systems have developed the control system (SCS[2]) that is in use on Bromma Conquips spreaders today. This system is used to steer the spreader and control that it performs correct. You can connect a PC to the master on an RS232[15] connection, on this connection you perform program download, the Spreader Program produced in ABE is send to the system in this way. The diagnostic of the system is also maintained through this connection.
BMS is the software that is used to handle both the download of programs and to monitor the status and diagnostic of the spreader system. The connection between the PC and the spreader system is only used when required.

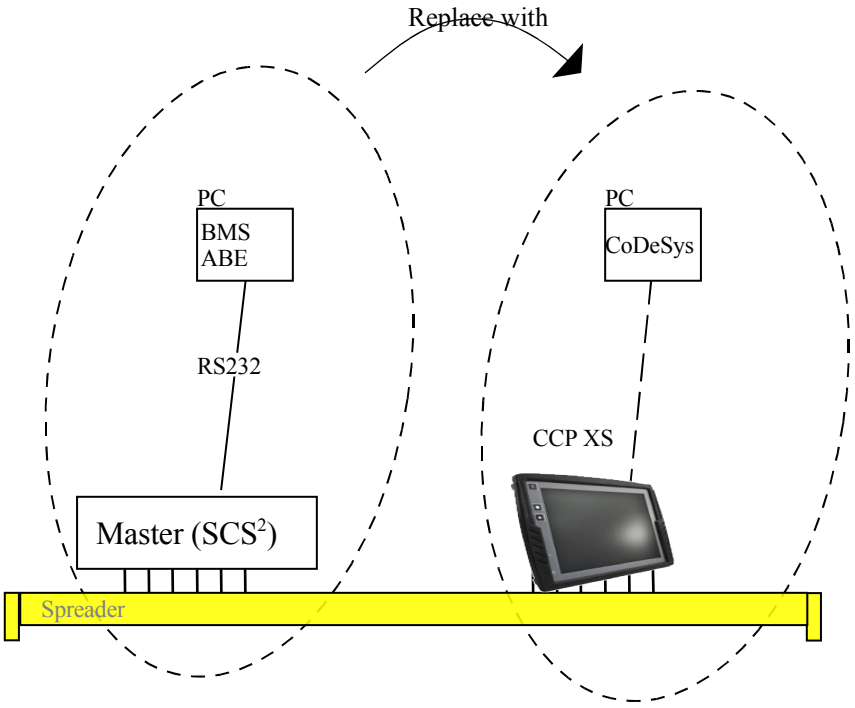


Figure 7: Showing the main idea being evaluated in the thesis

The main idea is to replace the Master computer in the current system with the on-board computer CCS XS (fig.7). The reason are the touch screen that can be used to directly monitor the system and that the Central Processing Unit (CPU) speed is higher leading to better performance of the spreader system.
In the Master you have the code for the different components, a component take input signal and according to the components specific logic calculate something and produce output signals. These components need to be ported and placed in the CCP XS and this concern is from now referenced as *porting of components*.

[15] Standard for serial binary data connection used in the computer serial ports

In the software tool ABE you decide which components to use and how to connect them and produce a description file that is downloaded to the master.
The expectation is to use CoDeSys when you design the component board in the new system. But there are many files of the format produced by ABE that is in use today in the current system, several of them large with over hundred connected components and the wish is to reuse them. The conversion of the ABE files to a file format that can be used in CoDeSys is from hereon referenced as *converting the application description*.

The handling of logging is in today's system maintained by BMS, an investigation is to take place about replacing it with tools provided by CoDeSys and is referenced as *replacing software for logging and diagnostics*.

3.1 Porting of components

When the System program in the SCS[2] is running it starts by reading the description file produced by ABE and builds a complete component board. The component board can be seen as the overall functionality of the system. Each specific component, for example compAnd (fig.8) is derived from the head class Component. When the instance is created all the ports of the component is also created, the component parameters are set and the component is stored in the componentBoard.

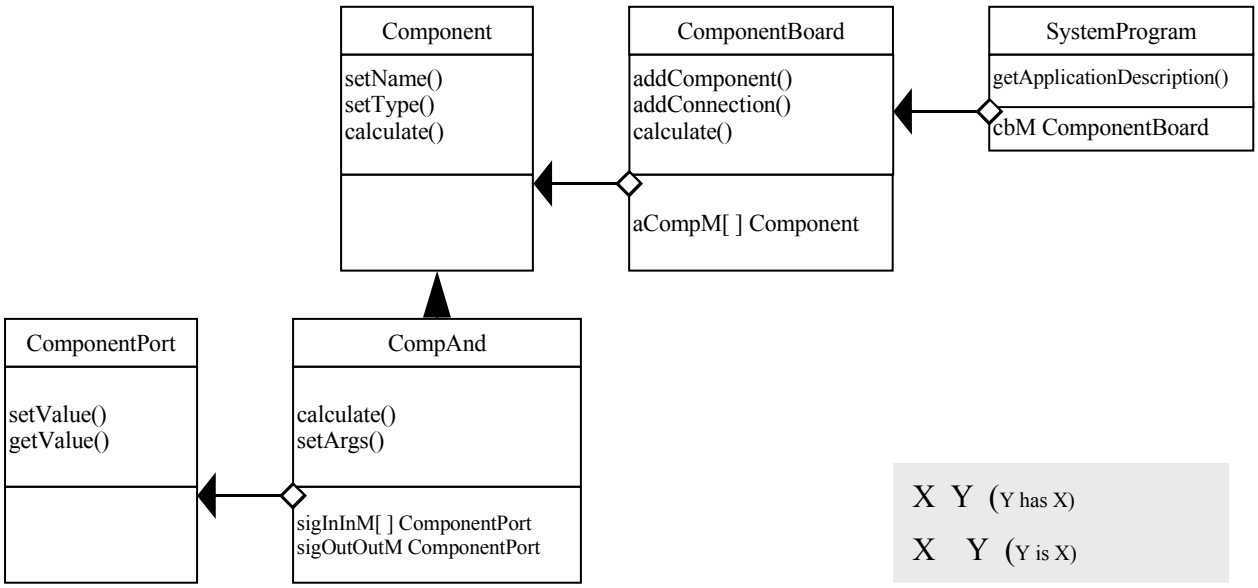


Figure 8: Class diagram showing the relations between the System Program and the Components.

When the complete component board is built the System program continuously calls calculate on the componentBoard. The componentBoard then calculates the output signals for all components depending on their inputs and logic

The components that are being connected in ABE exist precompiled in the target system. They consist of both ordinary Boolean logic gates such as: AND, NOT and OR gates but also as much more advance spreader specific components developed by Bromma Conquip. These components are complex and contain a lot of functionality and can also be altered individually by changing the component parameter setting.
Because of this; the complexity of many components but also that they have been tested and been in use for such a long time, it is desired that the porting of the components can be done with as little modification as possible. This is almost certainly most important for the specific code calculating the logic and setting output signals for each component.

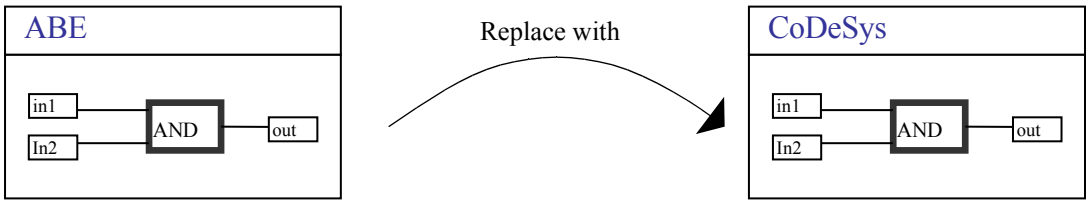


Figure 9: The building of applications is to be taken over by CoDeSys

In CoDeSys, we need to create the same thing in some way. The different components will have to be ported to a specific POU (fig.9). To build up the component board CoDeSys provide a number of graphical languages that probably could be used to connect the different components (in this case POUs).

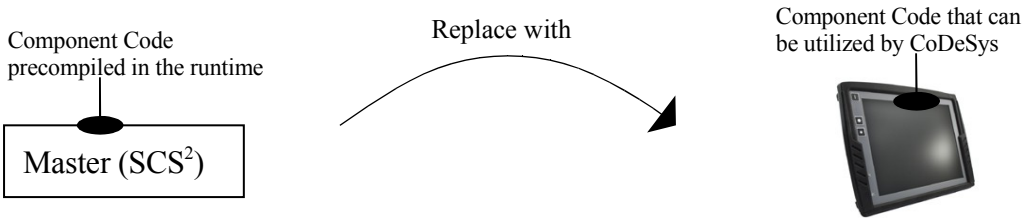


Figure 10: The code need to be ported in order to function in the new environment

The SCS² system was programmed in the object orient language C++. The components are programmed as objects in different classes (fig. 8) and the concern is how to best port the code for them (fig.10).
CoDeSys support all the languages in the IEC 61131-3 standard and the one that is most similar to the C++ code is Structured text, a text based higher layer programming language. But it is not object oriented and is more similar to plain Pascal- or C programming code. A piece of C-code would nevertheless have to be altered a lot to fit as Structured text in CoDeSys.
One other solution might be external linking. Since the runtime was implemented in ANCI C, it is possible and has been made possible by CoDeSys that provides an interface for linking user defined program code written in ANCI C into a program.

3.1.1 The setting of component parameters

When designing a component board in ABE it is possible to set a number of instance specific parameter. This is done as shown in the figure below (fig.11), the intent is that this also can be performed graphically when setting up your components in CoDeSys.

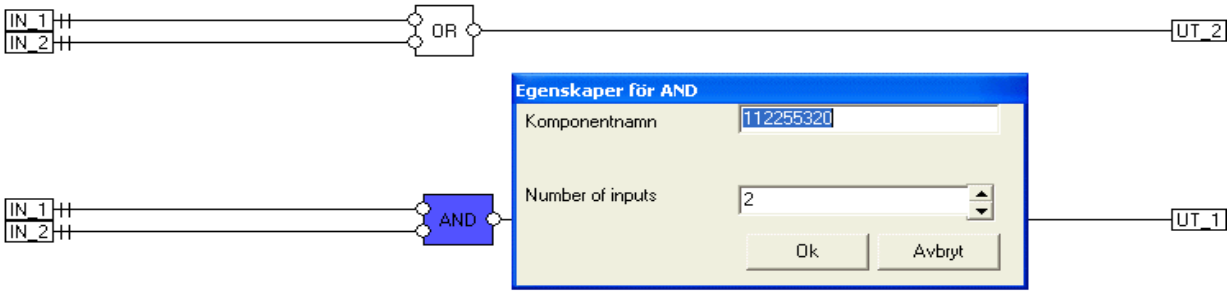


Figure 11: Component board in ABE, by right-clicking the AND component we can change its parameters.

There is no direct support for doing the same with the POUs in CoDeSys. So an alternative method has to be considered.

3.1.2 Change number of in/out- puts

As a result of changing the above mentioned parameters the number of inputs and outputs to the Component can be changed. CoDeSys have partial support for this when it comes to the operators that are included according to the standard IEC 611131-3: AND, OR, XOR and NOT. These components have the ability to change the number of inputs simply by right-clicking them and chose 'input of box'.
3S has been consulted about the ability to give this functionality to own defined components with a negative answer. If the number of inputs and outputs are fixed for a component we have to define the component for all possible number of ports.

3.2 Converting the Application description

The spreader program that is produced by ABE holds the application description.
Several Spreader Programs exist; many of them are complex and consist of a vast amount of components with connections. It would consequently take some time if all of them would have to be made again.

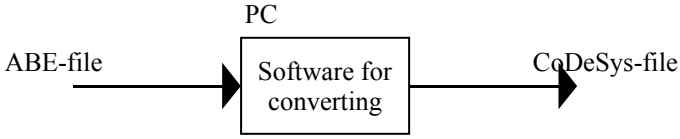


Figure 12: The idea is that the software tool reads an ABE file and transform its content and produce a CoDeSys file.

CoDeSys will be used when a new application description need to be designed. When we have decided how to best build a description file in CoDeSys we need to find a way to convert our old description files generated by ABE to a format that can be used in CoDeSys.
The format of the spreader program file consists of 4 different types: General information, Components, Connections and System information. The file is saved as a binary file (with the extension *spr). The purpose is to create a software tool that takes this file and depending on the information type, create one or several files that build up a corresponding project that can be utilized with CoDeSys.

3.3 Replacing system for logging and diagnostic

SCS^2 create a number of log lists: one for assertions and one error log also containing operation and system information:

- The assert log: when an assertion is found in the code the file and line where it happened is saved in the assertLog. This log list is of most interest to the developer of the system and not used for the system surveillance at operation.

- The error log: all components have access to the same variable (errorMessage) and use this to write messages about critical system behaviour, error and other general information to the log. The message is also displayed on the node. When the system needs diagnostic survey BMS is used to retrieve the log list.

These lists are handled in the same way. They are saved in the memory at the different nodes and can at need be loaded from the system with BMS and then be maintained on a PC.
The objective is to investigate if this method of handling the logging of critical events can be taken over by tools provided by CoDeSys.
CoDeSys provide two different tools for logging of events: Alarm configuration and Log:

- The Log tool is used to log events chosen from a list of four categories. These categories concern the operation of the CoDeSys program and can not be used for user specific actions that require that a log message should be created.

- The Alarm configuration tool is used to alert the operator of a critical state, record it and visualize it. The alert can be set off by a variable or an expression decided by the developer.

4 METHOD

In this section we discuss different methods that have been identified to solve the various parts of the thesis investigation. The methods are evaluated and the best for each specific problem is presented in the Solutions section.

4.1 Porting of components

These subheadings all correspond to different logic and visual problem that arose when porting the components.

4.1.1 Selecting POU to use for the components

In CoDeSys there are three different Program Organization Units (POUs) to use, if we are going to use them to represent the components we have some requirements:

- You need to be able to have several input and output signals to the component
- Save the parameter settings for the components
- You also need to be able to make different instances of a component.

The POU that doubtless fulfil these requirements best is Function Block, it is possible to make instances of them and we can represent each component in SCS[2] with its own Function Block. When making an instance of a component it will be given a specific name and its own variables, so it will be possible to set to the component specific parameters.

For some less complex components one other solution is possible and that is to use the function POU. This can be made for those components that do not have parameter settings, where all instances of a component are the same. CoDeSys provide some standard functions that could be used to replace some of Bromma's logical gates components.

4.1.2 Selecting language for the POUs

The best way to program the POUs is definitely to use the interface that CoDeSys provide and link in user defined functions written in ANCI C.
The other alternatives using CoDeSys own defined languages, with Structured Text being the best choice out of these, are no realistic alternatives. The main reason is that they have an own syntax for commands used in the language leading to that every line of code need to be changed. In the case of going from the object oriented language C++ to ANSI C the change is more in a structural way. Since C++ was derived from ANSI C the languages follow the same syntax patterns and the porting can be done without too many modifications.

CoDeSys call these users defined ANCI C functions for external functions and they have support for the POUs: Function and Function Block.

4.1.3 Selecting language for component board application

The goal with the component board application in CoDeSys is that it should function in a similar way as ABE. That you can build up component-scheme by choosing the components you wish and connect them to each other.
There are four graphical languages in CoDeSys:

- Ladder diagram, the ladder diagram is most suitable for creating program describing electric circuits.

- Sequential function chart, Sequential Function Chart is most suitable if the program can be divided into steps and run in chronological order.

- Continues function chart, it is very similar to FBD but has some additional features like feedback paths back directly to the POU and the possibility to decide the order of execution for the different components (fig. 14).

- Function block diagram is used to connect different Functions and Function Blocks, both standard and user defined. The signal flow always go from left to right in the diagram it is also divided in several networks to make it more perspicuous (fig.13).

The obvious reason for using the Function Block Diagram is that we are representing the components with Function Block POUs. ABE also use networks[16] when designing the component board. Furthermore, there is no interest in changing the order of execution for the different components, making the use of CFC unnecessary and in some case makes the system more prone to mistakes.

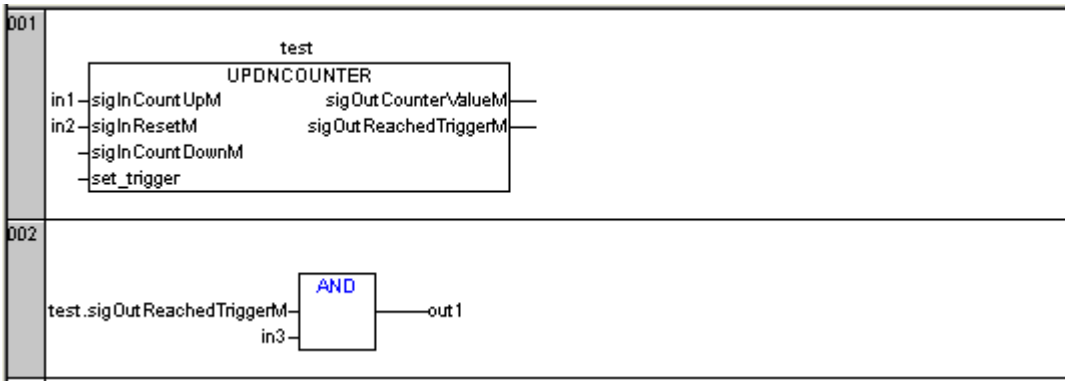


Figure 13: An example of a FBD with two networks.

There is nevertheless one big problem with FBD, it is not possible to decide in full how to place and connect the components in the different networks. If we look at the example in figure 13 we have already two networks and it is not possible to perform this connection using one network.

[16] In ABE the networks are known as rungs

This is due to the fact that we can not continue connecting components in the same network unless we use the most upper output, in this case the *sigOutCounterValueM* output. Even if we use it, we still need a new network for every lower output you connect to.
Some of the components that are in use on a spreader system from Bromma today have over 20 outputs. To use FBD with one of these components fully connected would lead to over 20 networks and the outline of the Component board would be difficult to grasp.
This rule for connections is just a fact and the recommendation from 3S regarding more complex systems is instead to use CFC.
The same scheme as in figure 13 would in use of CFC instead look as figure 14.

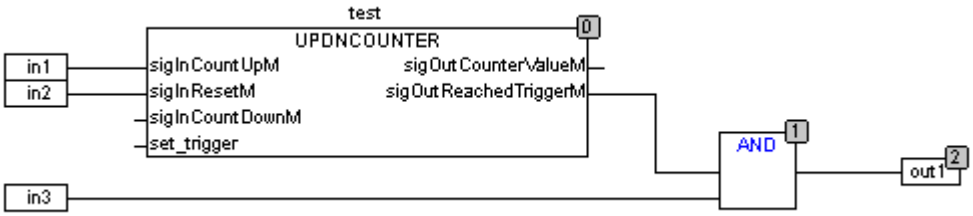


Figure 14: An example of a CFC

With CFC it is also possible to position the components in any way we want, with FBD the components is place automatically. One possibility if you don't wish the scheme to grow to far to the right is to use connection markers so that you can start from the left side again (fig.15).

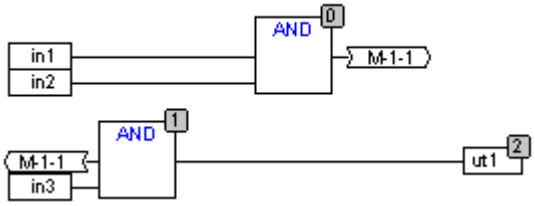


Figure 15: Showing the use of connection markers in CFC

4.1.4 The setting of component parameters

When designing the component board in ABE it is possible to alter specific information for the components.
The general way to do it in CoDeSys is to either set the internal variables for the Function Block when declaring it or to initiate the instances of the Function block when they are used in a program (fig. 16).

Figure 16: - To the left: initiation in the Function Block declaration
- To the right: initiate the internal variables for each instance of a FB

An additional solution is to declare a new POU used only for initiation. You could then graphically connect this POU to the component that is to be initiated. Then set each internal variable with an input box[17].
This last solution is the one we have chosen because of its graphical approach with similarities to ABE when the setting of parameter.

4.1.5 Change number of in/out- puts

For some of the components it is possible to change the interface directly in ABE by changing the parameter setting. For example could an AND component be set to have 2 or say 5 connected inputs.

We could use the included operator components that CoDeSys provide when possible. Those that are interesting are AND, OR, XOR and NOT. With NOT it is not possible to change the number of inputs by nature but it might as well be interesting to us instead of porting the component from the SCS^2 system. The rest of the components that is in use in the SCS^2 and will be ported have to adopt another way to handle the change in number of ports.

In CoDeSys each port is defined to a specific type and it is not possible to declare ports dynamically with us of arrays of a specific type. One method that we could us is to declare the component for all possible number of ports. This would also lead to separate code bases for all of them.

[17] For a better understanding see figure 23 in section 6.1.4

4.2 Converting the Application description

CoDeSys does not provide a standard way of representing different parts of a CoDeSys project. It is not possible to get a text representation of a project neither build one your self to input it for utilisation in CoDeSys.
CoDeSys provide a server called ENgineering Interface (ENI) server acting as a database system. It can be used for file sharing and version control when developing larger joint CoDeSys projects. When a CoDeSys program is connected to the database and project-files are put in the ENI, the information about different CoDeSys parts are presented in Extensible Markup Language (XML) representation and put in the database. When a project is fetched from the database and opened in CoDeSys it is converted back to a CoDeSys project.
If the Spreader Programs could be converted to the XML representation of the format that CoDeSys use, we could probably then make use of these old application descriptions in the new system.

4.3 Replacing system for logging and diagnostic

CoDeSys provide the tools: Log and alarm configuration.
Log is used to record events that happen which are preconfigured, for example state changes and user action.
With alarm configuration it is on the other hand possible to choose the expression that should trigger the alarm, it is then possible to both visualize and save a log corresponding to the alarm.

If the system for logging and diagnostic should be handled by CoDeSys and remain with the same functionality as in the SCS^2 system both tool can preferably be used:

- The Log tool can take over the role of the Operation log and handle spontaneous actions in the system.

- The Alarm Configuration tool can be used to handle the tasks that were handled by the Assert log and the Error log in the current system. Both Logs are triggered by a variable condition and the Error log is also producing an error message that is to be displayed.

The only real problem is how to make the linking between the error happening in the system and the alarm tool. A variable will probably be used in the runtime of the target but we need to make this variable visible for the CoDeSys application running on top.

5 SOLUTION

The methods presented here have been chosen as best solutions for the different isolated problems that have been analysed.

5.1 Porting of components

This section shows the whole solution for how a component is ported, including how to graphically place and connect components in CoDeSys and how the code from Bromma's old system has been altered in order to function with CoDeSys.

5.1.1 Function Block with External library

The editors in CoDeSys use a declaration part and a body. The declaration part holds all declared variables divided in input, output and local variables. The body holds text or graphic depending on which language you want to use.

The external ANCI C library can be managed in two different ways; either you put the ANCI C functions in the external library or directly in the run-time system. If we use the second way we only need to declare the different variables used by the component when building our Function Block. The code for the Function Block will be linked into the program at run-time.

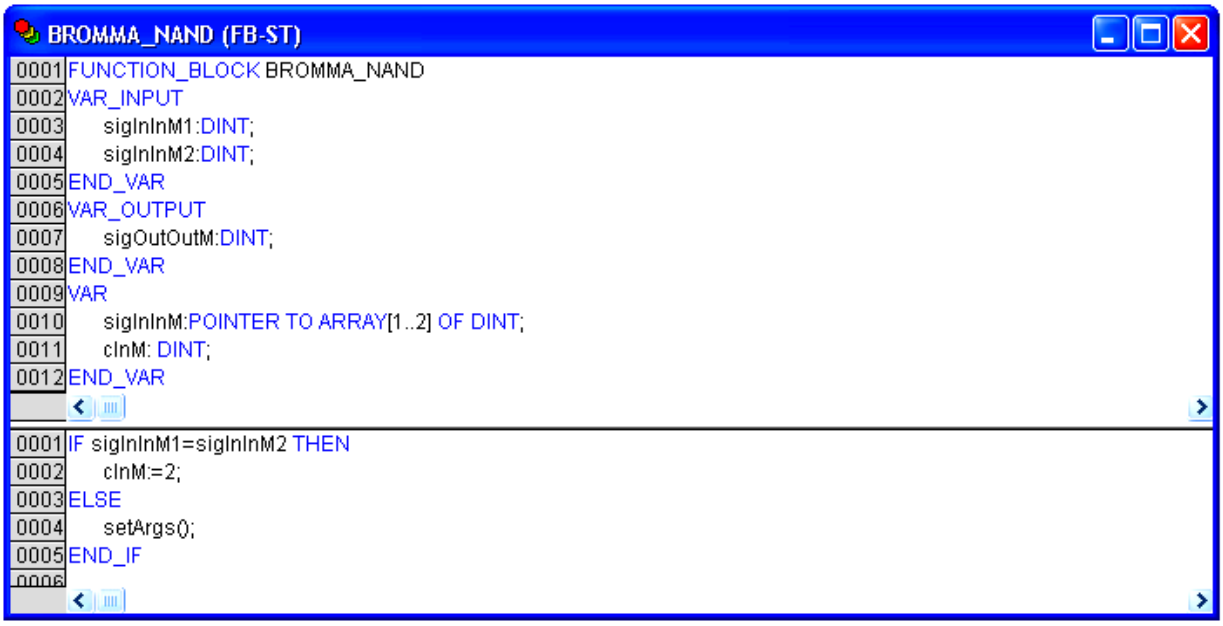


Figure 17: A Function Block representing a NAND component, the code in the body will be replaced at runtime.

When the interfaces for the components have been declared in the Function Blocks they are saved as an extern library, this lead to an automatic creation of a CoDeSys-library file and a C

header file.
The library file can then be opened and used by any CoDeSys program to include the different components in a project. The C header file is to be used in the C project creating the functionality for the Function blocks.

5.1.2 Implementing the library functions

The implementation of the different functionality for each component in the external library will be placed in an IO-driver. This driver is a way to add user specified adaptations to the runtime. In our case the IO-driver will have the extension DLL[18] depending on our platform being windows. This file is placed in the runtime and the functions will be linked dynamically when the CoDeSys program is downloaded to the PLC.
The IO-driver must also be registered in a configuration file in order for the runtime to recognize it.
The ANCI C code to the external functions is placed in a file called IODrvFunctions.c; this file holds the functions that make the IO-driver. The functions are called by the soft PLC as call-backs.
IODrvFunctions.c. also contain the reference table used when linking the function code in runtime to its interface.

5.1.3 Porting of the components C++ code into ANSI C

The C++ code for the components needs to be changed to ANCI C.
When we have saved our Function Blocks as an external library an ANCI C header file will be created. This file holds the definition for the different Function Blocks and is for the BOMMA_AND component created according to the figure below.

```
typedef struct
{
    DINT CINM;
    DINT SIGININM1;
    DINT SIGININM2;
    DINT SIGOUTOUTM;
}
BROMMA_ANDtyp;

/* The Prototypes */
void BROMMA_AND(BROMMA_ANDtyp* inst);
BOOL BROMMA_ANDinit(BROMMA_ANDtyp* inst, BOOL bRetain);
```

Figure 18: Header-file produced by CoDeSys when saving a Function Block as external library

We use the simple AND function to represent the idea behind external libraries in CoDeSys, the different parts of the header-file has the following purpose:

- BROMMA_ANDtyp: structure that will hold the variables of the specific Function Block. In this case it has two inputs one output and one internal variable, CINM.

[18] Dynamic Link Library: A Microsoft implementation for shared libraries

- BROMMA_ANDinit: this function is called upon initiation and is used if any variables need to be initialized.

- BROMMA_AND: this function is then called consciously during execution. Here the processing takes place; reading inputs, make calculations and write to the outgoing ports of the component.

These two functions need to be implemented and placed in the IO-driver file IODrvFunctions.c. This C-file will of course include the header file with the declarations of the functions and the structs.

If we now look at an example of how the code for a component was programmed in C++.

```
// CLASS:
//  CompAnd
//
// DESCRIPTION:
//  Implements the behavior of the logical AND function.
//  The component inherits from class Component and
//  overrides the methods 'calculate' and 'setArgs'.
//
class CompAnd : public Component
{
  public:
    CompAnd();  // Constructor
    ~CompAnd(); // Destructor
    void calculate();
    long setArgs(BlockParameter * parameters);
#ifdef CODEGENERATION
    char * generateDesriptionString();
#endif
  protected:
  private:
    long cInM;
    ComponentPort * sigInInM[20];
    ComponentPort * sigOutOutM;
};
#endif //COMPAND_H
```

```
// NAME:
//  calculate
//
// DESCRIPTION:
//  Calculates the output signals depending on
//  the input signals and internal state.
//
void CompAnd::calculate()
{
    long i;
    // Calculate the output signal
    for(i = 0; i < cInM; i++)
    {
        if(getValue(sigInInM[i]) == 0)
        {
            setValue(sigOutOutM, 0);
            return;
        }
    }
    // Set the output signal
    setValue(sigOutOutM, 1);
}
```

Figure 19: Definition of the AND component and the code for the member variable calculate.

As we saw in figure 6, compAnd inherits from the super class component. Many of the functions in the super class component and the specific components are not necessary to implement in the IO-driver since CoDeSys automatically handles it. It could for example be the member function setArgs in compAnd which set up the input ports and output ports of the component.
Instead it's particularly one function that is interesting: the calculate function. It is found in every component and is called by ComponentBoard (fig. 8), this function calculates the output signal depending on the input signals for every component.
This is the part we are interested in and that we want to port into ANSI C code and put in the IO-driver.

The Functions used by calculate is of course of the same interest.
The aim has been to keep the part inside the calculate function as unmodified as possible and if needed add code before and after to make this work. The reason is that the code has been tested a lot and been used in running applications for several years, to extensive changes could lead to that code parts have to be tested again.

5.1.3.1 Differences between C++ and ANSI C

The object-oriented language C++ is derived from the programming language ANCI C. The two languages have evolved independent of each other; this has lead to some further incompatibilities between the languages besides that one support the use of objects. The similarities are many. They are both imperative, sequentially processing a main loop, and the fundamental instructions for performing calculations and comparison follow the same syntax [22, 23].
The incompatibility that we need to worry about in our case is those that work in C++ but not in ANSI C [24]. A list of the most significant can be found here:

- Anonymous unions
- Classes
- Constructors and destructors
- Exceptions and try/catch blocks
- External function linkages (e.g., extern "C")
- Function overloading
- Member functions
- Namespaces
- New and delete operators and functions
- Operator overloading
- Reference types

The incompatibilities that can be found in our C++ code that is to be ported all relate to objects: classes, constructors/destructors, member functions/variables and new/delete operators.
These are the things we need to alter in the code to be able to port it into ANCI C and make it function together with CoDeSys.
The other incompatibilities mentioned above and other parts in C++, such as templates and the use of the vector or string class, is not present in the code we need to port and is therefore not discussed further.

5.1.3.2 The porting from C++ to ANCI C

What we have decided already is that a component is going to be represented as Function Block in CoDeSys and that we create a new instance of a component directly with a graphical language in CoDeSys. Therefore the operands to create and delete an instance will not be used. The same goes for the constructor that is called when a new instance of an object is created, the constructor often holds some initiation of member variables so this initiation needs to be moved some place else.
CoDeSys automatically generates an initiation function for all components that are declared in the external library (fig. 20). It is called once by CoDeSys at the beginning when a program is executing on the PLC. All initiation including the constructor initiation should be placed in this function.

```
typedef struct
{
        DINT CINM;
        DINT SIGININM1;
        DINT SIGININM2;
        DINT SIGOUTOUTM;
}
BROMMA_ANDtyp;

/* The Prototypes */
void BROMMA_AND(BROMMA_ANDtyp* inst);
BOOL BROMMA_ANDinit(BROMMA_ANDtyp* inst, BOOL bRetain);
```

Figure 20: Showing the declaration of structs and functions in ANSI C

The thing we need to focus on is the code representing the classes and the structure of a class (fig. 21). A class can hold functions and variables; these functions and variables are a part of the class and are called class members. When an instance is created from a class it can access its own member variables and can use its class functions to alter its variables.

```
// CLASS:
//  CompAnd
//
// DESCRIPTION:
//  Implements the behavior of the logical AND function.
//  The component inherits from class Component and
//  overrides the methods 'calculate' and 'setArgs'.
//
class CompAnd : public Component
{
  public:
    CompAnd();  // Constructor
    ~CompAnd(); // Destructor
    void calculate();
    long setArgs(BlockParameter * parameters);
#ifdef CODEGENERATION
    char * generateDesriptionString();
#endif
  protected:
  private:
    long clnM;
    ComponentPort * sigInInM[20];
    ComponentPort * sigOutOutM;
};
#endif //COMPAND_H
```

Figure 21: Showing the structure of a class in C++

The counterpart to the object oriented C++ class is in ANSI C the Structure abbreviated to Struct. The Struct is a way to combine simple data types into a complex one. Compared to the class it does not hold any functions only the variables types that that is part of the Struct. The format of the Struct is generated automatically when we produce an external library in CoDeSys (fig. 20). The member functions must instead be normal functions that can be called by anyone, this leads to some changes:

- The name of the function, if all components have a function called calculate that now is open for everyone to use, the program will not know which calculate belongs to which Struct.

- Variables in the function, a member function have direct access to the member variables.

The names of the functions have got to change, preferably to the name of the specific component represented by the class appended with the member-function name.

The declaration must also change and include an input parameter holding a pointer to the Struct with the variables for the component. One alternative could be to only have those member variables used by the function as input.

```
void CompAPS::calculate()
{
 .
 .

 currentSignalWindowM =
getCurrentSignalWindow(currentPosition);
 .

}
```

```
void APScalculate(APStyp* inst)
{
 .
 .

 inst->currentSignalWindowM =
APSgetCurrentSignalWindow(currentPosition, inst);
 .

}
```

Figure 22: - To the left, the member-function to the class CompAPS
- To the right the corresponding ported function in ANSI C

In the figure above the Automatic Positioning System (APS) component's memberfunction calculate have changed name to APScalculate, the same goes for the memberfunction(getCurentSignalWindow) that is accessed inside calculate. The Struct (inst) is also send into the function in order to access the variables of the component instance.

We have now decided so much that we can start a new CoDeSys project and make a Program POU, it must be named PLC_PRG and is always the main program.
We choose the programming language CFC and open the library manager. If we insert the library we made containing the Function Block interfaces we can now build a component board using these components. The code for the components are now changed and put in the PLC, when a program is downloaded the code is linked to the specific components used in the component board.

5.1.4 Changing component parameters

The method we have chosen is to declare a Function Block for changing the internal values of a component. The main reason is that this solution is more user-friendly when the user of CoDeSys graphically can see the possible variables that can be initialised.

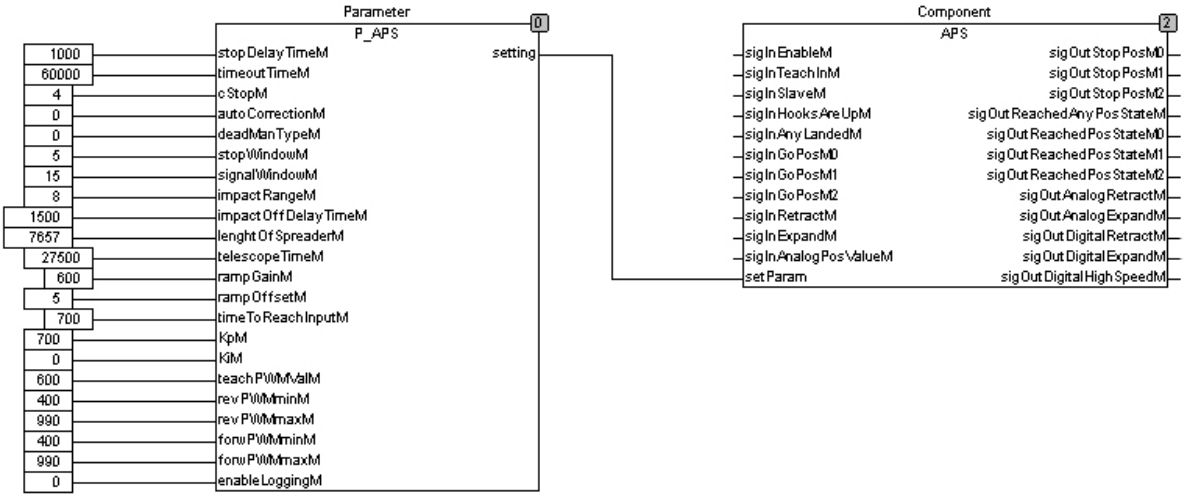


Figure 23: Shoving the idée with declaring a POU to set the internal variables of a component.

According to figure 23 you can now change the values in the input boxes (far to the left), the parameter POU will then read all inputs, put it in a structure that finally the component APS through the setParam input can read.
This is not a standard way for initiating parameters in CoDeSys and the initiation will not take place before the program is in running mode. Because of this a control can be made hindering the program to initiate the component with the same value every cycle.

5.1.5 Change number of in/out- puts

The method we have adopted is that we when possible use the CoDeSys provided functions that have inbuilt ability to change number of ports.
For the rest, meaning the most of the components, the solution will be to declare the components for all possible number of in- and out-puts as own Function Blocks
This means that they will all have an own structure, init-function and processing function. (See figure 20, section 5.1.3.2)
For complex Components this will lead to large amounts of code if they all would have individual code for the processing. The code that we are porting is the same for all possible interfaces for a component and is something we aim to keep.
The problem is that all variants of the component will be represented with an own Struct in ANSI C code and holding an extra input will lead to a larger Struct. But he ANSI C function receiving a Struct as an input parameter expects it to be of a certain size.
The solution is to make a general structure that hold all possible interfaces for a component and then have only one code part for the processing with this general structure as input. When a Component with a specific port collection is processed it has to be copied to this general structure and in that way access the code.

5.1.6 Retain variables

During the porting of the Bromma specific component APS another question arose. The Automatic Positioning System (APS) is used to teach the spreader a number of stop-positions.

They are saved in memory and are used to tell the spreader to extract or retract the arms to different stops. The stops are used to handle containers of different size.
Once these values are saved they should be retained even if the PLC is shut down.
In CoDeSys you declare these variables as RETAIN_VAR. In the IO-driver we can save our retain variables at need by calling the function IODrivReationSave(), the variables are then retained and will at next reboot replace the initiation value.

5.2 Converting the Application description

A software tool has been created that takes a CoDeSys file, that has been placed under database control in the ENI server that CoDeSys provide, then opens and traverses the ABE-file that need to be converted and place the information in the CoDeSys file.
Then the file is checked out from the database and the component board that was build in ABE can now be utilized in CoDeSys.
The driver to run the ENI server is licence free when used on a computers local file system, this serves our needs lovely.

The files placed under source control use a CoDeSys dependant file extension, but use XML representation inside the file to represent the components and connections used in the project file. The file holds some start information that informs that it is put under ENI control and also the size of the file, then the project information represented by XML syntax come finally after that an ENI specific file ending.
When the XML part of the file has been updated with information from the ABE file the whole file size is measured and the size part at the beginning of the file is updated.

The software tool created is as for the rest of the system only functioning for selected components. The tool has a function to position the components on the component board. The spreader file from ABE holds information about the position for the components, but they need to be altered because of difference in measurement between ABE and CoDeSys. The visual size of a Component when placed in the component board also differs between the two. The CoDeSys solution that was closest to hand in the beginning was Function Block Diagram where the outline is stricter and the positioning is automatic. In the Continuous Function Chart the disposition is free and all Functions and Function Blocks used have a position. This has lead to a solution for positioning that was not considered during design phase, instead it was add on to the already existing software.

The additional work that needs to be done is to update the tool so that it can handle all available components.
The software is currently not working for ABE spreader files divided in several networks, CFC is not using networks but the chart might be to easier follow if it was divided in a network manner, this could in that case be done with the use of connection markers. (fig. 15)

The design for parts of the program might need revising because of a late change of graphical language. If FBD had been chosen the positioning of components in CoDeSys would been made automatically. Now with the choice of using CFC, all parts on the component board need a 2 dimensional position.

The parts of the software tool for converting that is not complete:

- It needs to be updated to handle all available components
- It needs additional parts to handle larger application files that use networks
- Calculating and positioning(x and y coordinates) in the component board

5.2.1 Name of Converter Demonstrator

One software tool has been developed, the ABE to CoDeSys converter. It is programmed to show and verify the theory of the "*converting the application description"* part of the thesis.
It is programmed in C++ and the interface is developed with the MFC application Wizard[19].

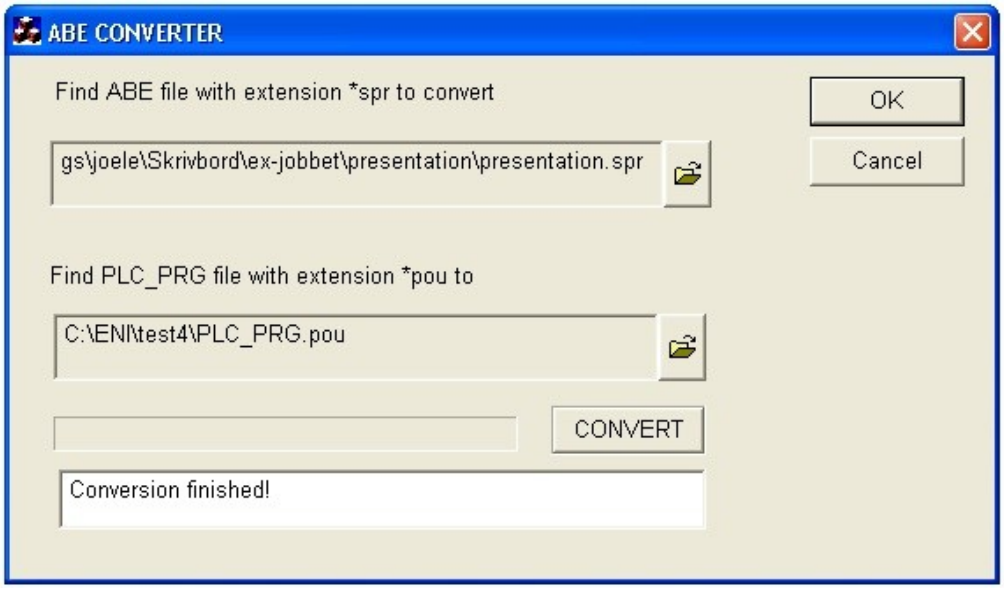


Figure 24: The Graphical interface for the ABE to CoDeSys converter

In appendix 1 you will find a 'quick start guide' for using the software tool and the necessary steps that need to be taken in order for a complete transaction of an ABE application.

5.2.2 Pseudo code for the converting tool

Pseudo code is a simplified variant of the implemented C++ code that mixes ordinary words with the most common computer words.

Table 1

Description of the computer terms used in the pseudo code	

{
Lines of code
}
	The lines of code inside the brackets are executed depending on the statement before the starting bracket ({).

[19] A help for developing computer programs using the Microsoft Foundation Classes

While(x)	Repeat the code inside the brackets as long as condition x is true
If(x)	If condition x is true then once execute the code in the brackets.
else	If an If(x) condition is false then an else condition can take alternative actions

For further understanding of the pseudo code please see section 3.1.1 about the structure of the ABE file. See also appendix 1 for a complete guide that explains the steps that need to be performed before the converting tool can be used.

Pseudo code

```
Load the ABE file (.spr)
While (ABE file not empty)
{
    Read the next word/type
    If (Next type = GENERAL)
    {
        Read all GENERAL information and save useful parts
    }
    If (Next type = SYSTEM)
    {
        Read all SYSTEM information and save useful parts
    }
    If (Next type = COMPONENT)
    {
        Read how many parameters the component has
        Save it temporary in a variable: number_of_parameters
        Read what type of component it is
        Save the component as an object in a list.
        Give the component an id
        Read and save the name of the component
        If (type of component = input or output)
        {
            Save the element type as:  input or output
            Read and save the type as: analogue or digital
        }
        Else
        {
            Save the element type as a box component
            Save the type of the specific component
            While(loop number_of_parameters times)
            {
                Read and save parameter value
            }
        }
    }
    If (Next type = CONNECTION)
    {
        Read the connections source component
        Read the port name of the source component
        Find the id of the source component in the list of component objects
        Read the connections destination component
        Read the port name of the destination component
        Find the destination component in the list of component objects
```

```
        Save id and port name for the source component in the destination component object
    }
}
Load the CoDeSys file (.pou)
(Write the information saved in the list of component objects to the CoDeSys file )
Write to declare all common components
Write to declare all input components
Write to declare all output components
While (until the list of component objects is traversed)
{
    Write the element type of the component to the file
    Write the id of the component to the file
    Write the name of the component to the file
    If (It is common components)
    {
        Write the type of the component to the file
    }
    For (all ports with destination connections)
    {
        Write id and port for the component being the source of the connection
    }
}
```

For an example of how the CoDeSys (.pou) file could look like after running the converter tool, see appendix 3.

5.3 Replacing system for logging and diagnostic

The best solution for replacing the current logging and diagnostic with CoDeSys provided tools is to use both the Alarm Configuration and Log tool.

The Log tool will take the functionality that the Operation Log was performing, this log can be monitored directly during or after the online section to trace errors.

The two other logs used in the SCS^2 are preferably handled by CoDeSys Alarm Configuration where you can decide more precisely what to Log and how to do it. It also has a build in visual function for displaying the alarm that has set off, we can also decide the importance of the alarm. The importance then leads to different actions that have to take place for the operator that receives the alarm.

The biggest problem with this solution is probably how to reach the variables that start off the alarm, if we use the same solution as Bromma did in the SCS2 system, a set of global variables could be used to manage these events. The variables would then be together with the code that lie in the runtime and that is linked when the softPLC is started. But we need to reach the alarm variables in the runtime so that they can be visualized.

CoDeSys has an own Class called PLCHandler that can be used for communication between clients and a PLC, in our case the client could be the visualization. With the Handler we could then cyclically read our global alarm variables that we use in our PLC and communicate them to the visualization client that interpret the variables and alarm the operator if something is going wrong.
But this is not a walk in the park; the PLCHandler is complex and probably not needed for this simple operation. The PLCHandler is not for free, the cost is not motivated if its use is to only communicate a couple of variables.
The solution could instead be to define a new extern function with a global variable that holds the different alarms. A function is defined together with the struct in the same way as Figure 17 and the definition and implementation can be placed together with the code for the components. The function could then send the address of the global structure variable and the structure can be updated with any alarm events that is caused while processing the plc program.
In sending the address the actual variable will be updated once the address pointer receives the new alarm value and can then show the result in the Alarm Table in the Visualization.

5.4 Conclusion of solution

The porting of Bromma Conquips software components to CoDeSys will be made using:

- External library holding Function Blocks that represent the components, the code for the FB is written in ANSI C and is linked at runtime.

- In the main program in CoDeSys we will then use a Continues Function Chart to build up the component board using the Function block from the external library together with standard functions provided by CoDeSys.

- The ENI server will be used to convert an existing application description build in ABE to one that can be utilized in CoDeSys

- Both The Log and Alarm tool are suitable to take over different loggings used in SCS^2, with the alarm tool and its alarm table errors occurring can be displayed directly on the CCP XS.

6 TESTING

At the end of the thesis period a test was performed to show that the concept, the porting from the old system to the one that is based on CCP XS and CoDeSys, was functioning.
The specific component that was tested was the automatic positioning system (APS) that is used to position the telescope arms. The component is used to first teach the arms a number of positions that the arms retract and extract to for different size of containers. When the spreader has been thought these stops it is for example possible to tell it to go to the 40 feet position, this could for the figure below (fig. 25) be to set a Boolean true value to input sigInGoPosM2.

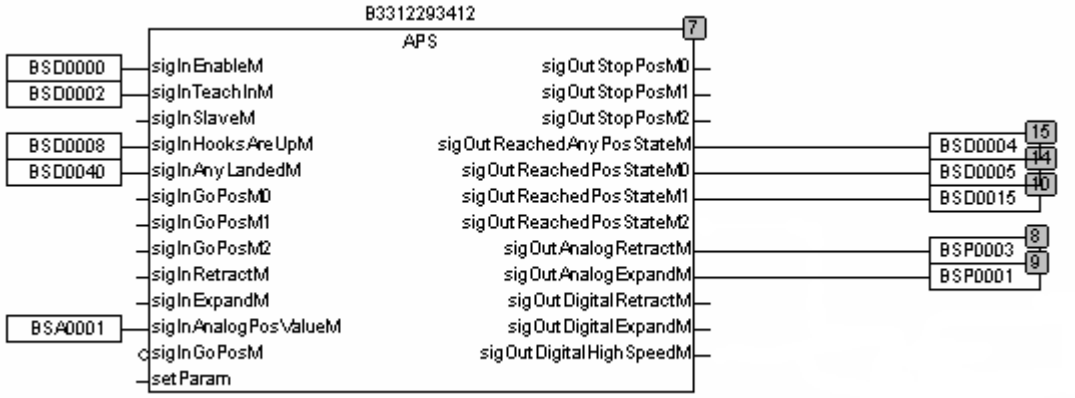


Figure 25: Shoving the graphical interface for the APS component in CoDeSys

The APS was chosen because it is one of the most complex components used for controlling the spreader. Visually it is also easy to see if it performs correct when it is positioning the arms of the spreader by extracting and retracting them.

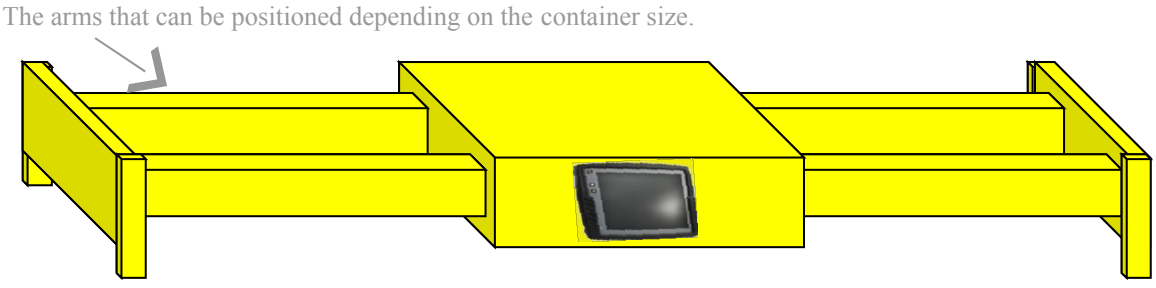


Figure 26: Showing a spreader mounted with a CCP XS

The ported component was placed in the CCP XS, now connected to control the entire spreader replacing the old master (fig. 26). A small program was made in CoDeSys that connected a number of input sensors, the APS and a number of output actuators. One actuator is responsible for making the arms extract them self.
The test became successful, the APS performed as expected which shows that the main investigation in the thesis was fulfilled. The second part of the test was to see if the replacement

is worth it in respect to performance.

6.1 Performance Test

During the test a number of performance data was collected. These data was more a control that the system actually is performing as expected, the CCP XS is more powerful than the master used on the spreader today so instead of a comparison between the systems it is more an insurance that the system is performing according to expectations.

The data that was of most interest to us was the cycle time for the component, the APS was chosen for the test because it is close to overloading the system when it is in use today. So if this component performs well in the new environment then Bromma's other components most likely will too.

Table 2

APS	RUNNING			IDLE		
	min	average	max	min	average	max
1. SCS2 (16MHz)	38ms	**39ms**	48ms	-	-	-
2. CCP XS (533MHz)	96µs	**118µs**	138µs	71µs	**92µs**	117µs
3. CCP XS (533MHz)	115µs	**150µs**	171µs	85µs	**110µs**	128µs

1. The current system loaded with the APS and during a simulation going to different positions.
2. The CCP XS loaded with the APS and cycle time recorded both when going to a position (running) and while waiting(idle)
3. The CCP XS loaded with a modified APS that is more general in handling different number of ports, discussed in section 6.1.5

As seen in table 2 we didn't make any recording of values for the old system at idle mode, this was due to the fact that the SCS2 system only gave us cycle time information twice every hour. And when we ran out of time we decided to test it while running.

The result is positive with the performance for the APS being up to 300 times faster: 0,118 ms for the CCP XS compared to 39 ms in the current system.
The result is due to the faster and more modern CPU[20] in the CCP XS. Only the CPU speed is 30 times faster: 533 MHz compared to 16 MHz. Further reasons for the low cycle time is probably that the newer CPU is equipped larger cache memories and use pipelining in the processor might lead to lower cycle time [25, 26, 27].

We also performed a test to see if the solution for the parameter setting altered the cycle time in

[20] The CCP XS use an Intel Xscale processor (533 MHz, 32+32 KB instruction and data cache, 133 MHZ bus speed)

The SCS[2] use a Motorola 68332 processor (16 MHz, 2KB static ram, 16 MHz bus speed)

any way, but the result was negative. The result was expected because the inputs to set the parameters are static and can not be changed online. They are recorded once and when the parameters have been initialized these inputs are not read again and therefore should not affect the performance.
Another thing worth mention is that the current system runs with an interval of 50 ms and with a cycle for the APS at average 39 ms the system probably experience problems with jitter sometimes. This should not be a problem now. The CPU utilization was also recorded for the new system giving 6 % utilization only for the PLC, with the APS loaded but idle it was 7% and while running it came up to 8% of CPU usage. So the porting to the new system leads to lots of spare time to use for further control and improvements of the system.

6.2 Size of ported code

The porting of the components from the old system has lead to changes of the code.
The original code, written in C++ had to be changed into ANSI C code to be able to function with CoDeSys. This has lead to modifications in the code and the size of each component has changed.
The biggest effect has nevertheless the problem concerning the change in number of ports[21].
We can also see that the general code that is programmed to use the same component code despite the number of inputs is performing slower: 150 µs compared to 118 µs is an increase of 27 %.
Further evaluation is probably needed before deciding about the best solution.

[21] See section 5.1.5

6.3 Efficiency of produced code

CoDeSys produce a binary file containing the PLC program that is downloaded to the target. This file includes the interface for those function blocks that lay precompiled in the target. The Comparison between the produced codes generated from the CoDeSys program containing CFC with one produced using IL (Instruction list) can give us clue of how efficient the CFC code is produced.

The example code we use to conduct this example is very basic. In the first test two input an AND gate and one output is used, in the second test we added an AND gate and one more output.

Table 3

	1 AND gate, 2 inputs, 1 output	2 AND gates, 3 inputs, 1 output
CFC	21,2 kB (21 808 byte)	21,5 kB (22 082 byte)
IL (from CFC)	21,2 kB (21 808 byte)	21,5 kB (22 088 byte)
IL	21,2 kB (21 772 byte)	21,3 kB (21 898 byte)

In the test the size of the file produced from CFC and IL that has been converted in CoDeSys to IL is basically the same. The difference comes when we produce the IL our self, the file size is here a bit smaller. This is depending on that we know where we do not need to use temporary variables to store the outcome from a function, if the result is to be used at once in the next calculation we do not need to store it and reload it, it is already loaded and can be used.
This case of storing and reloading is exactly what happens in the IL file converted by CoDeSys and by comparing the file size the CFC file used to download is probably produced in the same way.
In a larger system with many components, temporary variables are nevertheless necessary and the efficiency of CFC will be close to code produce in IL.

In our system the most of the code will be ported and put in libraries to be linked at runtime and is therefore not of impact for this test, the test can nevertheless have some interest because of the interface description and standard function that we have chosen to replace some of Brommas components. From the 3s homepage we found that the ANCI C code generation is roughly 20 to 30 % slower than for Instruction list [28].

7 Discussion and Conclusions (Survey)

This thesis has showed that it's possible to perform the main task that has been evaluated in the thesis; it is possible to port the entire system that is in use today on Bromma Conquips spreaders to use it with CoDeSys on the on-board computer CCP XS executing a soft PLC.

The final test that was performed at site at one of Bromma Conquips factories gave positive result but at the same time something to think about. The performance data showed that some of the solutions, especially how to handle dynamic ports, might have to be considered and maybe re-evaluated to see if an alternative solution is necessary. The test for the whole picture was a success, the concept hold and the performance increase for the system when running its most consuming component was very good. But what might need some revising and further investigation is how to handle the dynamic part of the components in the new system. With dynamic part referring to the ability to alter the number of in- and out-puts. But considering how much lower the cycle time became this could willingly be afforded, it is of now reason to doubt the overall improvement the using of the CCP XS for steering the spreader would be.

I can conclude by saying that the thesis work has been interesting and the fact that it has been so closely connected to the industry has made it a good experience and start to my working career.

8 REFERENCES

[1] CC Systems, nov 2005
http://www.cc-systems.com/

[2] CC Systems AB, ”*Exjobbsbeskrivning - CoDeSys på CCP XS för Bromma Conquip”,* Document

[3] Knut Sjöholm och Fredrik Löwenhielm, ”Mjukvarubaserad PLC-tolk för IEC 61131-3”, Uppsala University 2001

[4] Mooney, J.D. “Strategies for supporting Application Portability”, IEEE Computer 23, 11 (Nov, 1990), 59-70

[5] T. Tanaka, M. Hakuta, N. Iwata, M. Ohminami, "Approaches to making software porting more productive," *tron*, p. 73, The 12th TRON Project International Symposium, 1995, 1995.

[6] William Edward Woody, “Strategies for Porting Software between Microsoft Windows and Apple MacOs”, jan 2002
http://www.chaosinmotion.com

[7] Mooney, J.D. “Bringing Portability to the Software Process”, Technical Report TR 97-1, West Virginia University, Dept. of Statistics and Comp.Sci., 1997.

[8] Krueger C.W. “Software reuse”, ACM Computing surveys, volume 24, issue 2, 1992

[9] Microsoft .NET Homepage, may 2006
http://www.microsoft.com/net/default.mspx

[10] IEEE Technical Council on Software Engineering, may 2006
http://frakes.cs.vt.edu/tcsereuse.html

[11] Home — software reuse, may 2006
http://softwarereuse.nasa.gov/

[12] “Controller Area Network (CAN), an overview”, Controller Area Network (CAN) - CAN in Automation (CiA), Jun 2006
http://www.can-cia.org/can/

[13] Andy Lewis, Bromma Conquips AB, “Spreader Control System- 2: nd generation SCS²”, Internal Document

[14] Hans Svanfeldt, CC-Systems AB, “ABE User Manual”, version 1.0 - revision PB5, Manual

[15] Hans Svanfeldt, CC-Systems AB , "Spreader Program format SCS²", Internal Document

[16] Hans Svanfeldt, CC-Systems AB, "User Manual BMS – Light", version 1.0 – revision D, Manual

[17] "IEC 61131-3, 2nd Ed., Programmable Controllers Programming Languages", International Electrotechnical Commission, 2001

[18] Eelco van der Wal, "Introduction into IEC 1131-3 and PLCopen", 1999

[19] "User Manual for PLC Programming with CoDeSys 2.3", version 3.6, by 3S – Smart Software Solutions GmbH

[20] "CoDeSys SP 32 Bit Full", version 0.8, by 3S – Smart Software Solutions GmbH

[21] "The CoDeSys Business Model", 3S Smart Software Solutions, mar 2006
http://www.3s-software.com

[22] "C programming language", Wikipedia, the free encyclopedia, jan 2006
http://en.wikipedia.org/wiki/C_programming_language

[23] "C++", Wikipedia, the free encyclopedia, feb 2006
http://en.wikipedia.org/wiki/C%2B%2B

[24] "Incompatibilities Between ISO C and ISO C++" by David R. Tribble, jan 2006
http://www.david.tribble.com/text/cdiffs.htm

[25] "Instruction pipeline", Wikipedia, the free encyclopedia, apr 2006
http://en.wikipedia.org/wiki/Pipelining

[26] "MC68332 Product Summary Page", Freescale semiconductors, may 2006
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC68332&nodeId=0162468rH3YTLC61650795

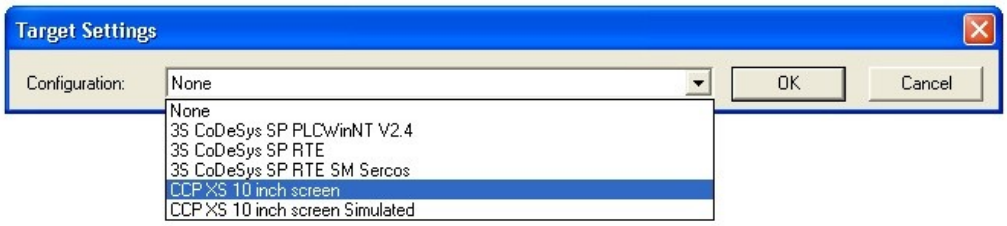
[27] "Intel® IXC1100 Control Plane Processor", Intel corporation, may 2006
http://www.intel.com/design/network/products/cpp/ixc1100.htm#features

[28] "Frequently Asked Questions", 3S Smart Software Solutions, mar 2006
http://www.3s-software.com/index.shtml?CoDeSys_faq&_printversion#cycle

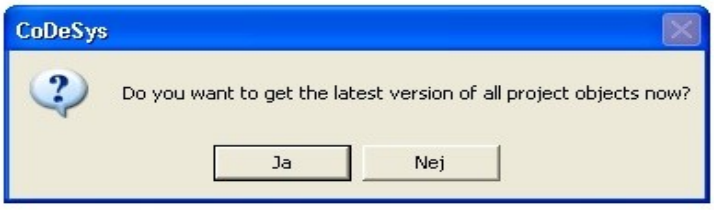
Appendix 1 Quick start guide

Manual for demonstrator 'ABE to CoDeSys converter':

1. Open the programming tool CoDeSys.
2. Chose file->new and select target setting: CCP XS 10 inch screen.



3. In New POU select program and CFC, then press ok.
4. Then chose project-> options -> database connections and check: 'use source control (ENI)'
5. In the same window, press 'configure ENI …'
6. Here you define a project name, this name will be the catalogue under C:\eni where your CoDeSys object will be placed.
7. When this is done, press next and enter your database username and password (see manual for ENI admin).
8. project-> project database-> multiple define. Chose project and select PLC_PRG.
9. The PLC_PRG is now in the database, close and save the project.
10. Open the explorer: C:\eni\"project name" right click PLC_PRG and uncheck the write protection under properties.
11. Open 'ABE to CoDeSys converter' and select ABE file to convert and the CoDeSys file PLC_PRG in C:\eni\"project name". Then press 'convert'.
12. Open your CoDeSys project again and in the automatically opened window chose 'get latest version of project objects now?'.



13. And your ABE application description is now available in a CoDeSys environment.

APPENDIX 2 TERMS OCH DEFINITIONS

Word/Abbr.	Description
ABE	Application Builder Environment, application to program the SCS^2
ANSI	American National Standards Institute
API	Application Programmatic Interface, provides services that can be used by other computer programs.
APS	Automatic Positioning System, Bromma specific component used to teach and steer the spreader to different positions depending on container size.
BMS	Bromma Monitoring system, tool to supervise scs^2
CAN	Controller Area Network, multi master network that connects several electrical units.
CCP XS	Cross Country Pilot XS, A compact, robust and versatile on-board computer for Windows CE or Linux. Developed by CC-Systems. XS is referring to the size.
CFC	Continues Function Chart
CiA	CAN in Automation, organisation for developers and users.
Class	It is on object oriented programming languages used to group variables and functions that relate to each other.
CoDeSys	**Co**ntroller **De**velopment **S**ystem, programming environment to make control applications.
Component board	Holds information about which components and ports that is used and how they are connected in the system.
CPU	Central Processing Unit, interprets instructions and calculate data in a computer
DLL	Dynamic Link Library, holds a number of programs that can be called on and loaded at need by the executing file.
ENI server	CoDeSys **En**gineering **I**nterface, used for version control and project cooperation
FBD	Function Block Diagram
IEC	International Electrotechnical Commission
IEC 61131-3	Open global PLC programming standard
IL	Instruction List
Instance	Member of a class used in object oriented programming languages.
Master node	Collect input from other nodes performs calculation and send out new outputs
MFC	Microsoft foundation Classes
PLC	Programmable Logic Controller, used to supervision and control processes. It reads inputs and depending on logic sets the outputs.
PLCHandler	Make communication between PLC and a client possible
POU	Program Organising Unit
Pseudo code	Code that mix ordinary words with the most common computer words
RS232	Standard for serial binary data connection used in the computer serial ports
SCS^2	Spreader Communications System (generation 2)
Slave node	Send input values to the Master and set output values according to received data from the Master.
Struct	A way to combine simple data types into more complex ones
XML	Extensible Markup Language

APPENDIX 3 CODESYS FILE IN XML REPRESENTATION

Example of a CoDeSys file put in the ENI server, all information about the specific file is represented in XML notation. This is at the same time an example of how a file produced by the 'ABE to CoDeSys converter' tool could look like.

```
_ENI_ ¤ <?xml version="1.0" encoding="ISO-8859-1"?>

<pou>
    <path/>
    <name>PLC_PRG</name>
    <flags>2048</flags>
    <interface>
        <![CDATA[PROGRAM PLC_PRG
VAR
    B612492628: FAC;
    B3312293412: APS;
END_VAR
VAR_INPUT
    BSA0001: DINT;
    BSD0040: BOOL;
    BSD0008: BOOL;
    BSD0002: BOOL;
    BSD0001: BOOL;
    BSD0000: BOOL;
END_VAR
VAR_OUTPUT
    BSP0003: DINT;
    BSP0001: DINT;
    BSD0015: BOOL;
    BSD0013: BOOL;
    BSD0007: BOOL;
    BSD0005: BOOL;
    BSD0004: BOOL;
END_VAR]]>
    </interface>
    <cfc>
        <element type="box">
            <id>16</id>
            <position>56,12</position>
            <isinst>true</isinst>
            <text>B612492628</text>
            <type>FAC</type>
            <input inputid="0" inputidx="0"/>
            <input inputid="15" inputidx="11"/>
            <input inputid="15" inputidx="6"/>
            <input inputid="0" inputidx="0"/>
            <output/>
            <output/>
            <output/>
```

```
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
        </element>
        <element type="box">
            <id>15</id>
            <position>26,1</position>
            <isinst>true</isinst>
            <text>B3312293412</text>
            <type>APS</type>
            <input inputid="1" inputidx="0"/>
            <input inputid="2" inputidx="0"/>
            <input inputid="3" inputidx="0"/>
            <input inputid="7" inputidx="0"/>
            <input inputid="11" inputidx="0"/>
            <input inputid="0" inputidx="0"/>
            <input inputid="8" inputidx="0"/>
            <input inputid="0" inputidx="0"/>
            <input inputid="0" inputidx="0"/>
            <input inputid="0" inputidx="0"/>
            <input inputid="12" inputidx="0"/>
            <input inputid="0" inputidx="0"/>
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
            <output/>
        </element>
        <element type="output">
            <id>14</id>
            <position>56,10</position>
            <text>BSP0003</text>
            <inputid>15</inputid>
            <inputidx>7</inputidx>
            <negate>false</negate>
        </element>
        <element type="output">
            <id>13</id>
            <position>56,11</position>
            <text>BSP0001</text>
            <inputid>15</inputid>
            <inputidx>8</inputidx>
            <negate>false</negate>
        </element>
        <element type="input">
            <id>12</id>
            <position>6,13</position>
            <text>BSA0001</text>
```

```
            <negate>false</negate>
        </element>
        <element type="input">
            <id>11</id>
            <position>6,7</position>
            <text>BSD0040</text>
            <negate>false</negate>
        </element>
        <element type="output">
            <id>10</id>
            <position>56,8</position>
            <text>BSD0015</text>
            <inputid>15</inputid>
            <inputidx>5</inputidx>
            <negate>false</negate>
        </element>
        <element type="output">
            <id>9</id>
            <position>76,24</position>
            <text>BSD0013</text>
            <inputid>16</inputid>
            <inputidx>7</inputidx>
            <negate>false</negate>
        </element>
        <element type="input">
            <id>7</id>
            <position>6,6</position>
            <text>BSD0008</text>
            <negate>false</negate>
        </element>
        <element type="output">
            <id>6</id>
            <position>106,14</position>
            <text>BSD0007</text>
            <inputid>16</inputid>
            <inputidx>0</inputidx>
            <negate>false</negate>
        </element>
        <element type="output">
            <id>5</id>
            <position>56,7</position>
            <text>BSD0005</text>
            <inputid>15</inputid>
            <inputidx>4</inputidx>
            <negate>false</negate>
        </element>
        <element type="output">
            <id>4</id>
            <position>56,6</position>
            <text>BSD0004</text>
            <inputid>15</inputid>
            <inputidx>3</inputidx>
            <negate>false</negate>
        </element>
        <element type="input">
            <id>3</id>
            <position>6,4</position>
```

```
            <text>BSD0002</text>
            <negate>false</negate>
        </element>
        <element type="input">
            <id>2</id>
            <position>6,39</position>
            <text>BSD0001</text>
            <negate>false</negate>
        </element>
        <element type="input">
            <id>1</id>
            <position>6,3</position>
            <text>BSD0000</text>
            <negate>false</negate>
        </element>
    </cfc>
</pou> _ENI_ joel
```